Experiment No.6
Process Management: Synchronization a. Write a C program to implement the solution of the Producer consumer problem through Semaphore.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Write a C program to implement solution of Producer consumer problem through Semaphore

Objective:

Solve the producer consumer problem based on semaphore

Theory:

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer. Producer consumer problem is a classical synchronization problem.

A **semaphore** S is an integer variable that can be accessed only through two standard operations : wait() and signal().
The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

```
wait(S){

while(S<=0); // busy waiting

S--;

}

signal(S){

S++;

}
```

To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Initialization of semaphores –
mutex = 1
Full = 0// Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

Solution for Producer –

```
do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer


signal(mutex);

signal(full);

}while(true)
```

When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of "full" is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Solution for Consumer –

do{ wait(ful

l);

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

Program:

```
// C program for the above approach
#include <stdio.h>
#include <stdlib.h>

// Initialize a mutex to 1
int mutex = 1;

// Number of full slots as 0
int full = 0;

// Number of empty slots as size
// of buffer
int empty = 10, x = 0;
```

```
// Function to produce an item and
// add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;

    // Increase the number of full
    // slots by 1
    ++full;

    // Decrease the number of empty
    // slots by 1
    --empty;

    // Item produced
    x++;
    printf("\nProducer produces"
           "item %d",
           x);

    // Increase mutex value by 1
    ++mutex;
}

// Function to consume an item and
// remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;

    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes "
           "item %d",
           x);
    x--;

    // Increase mutex value by 1
    ++mutex;
```

```
}

// Driver Code
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'
// can  give wrong value due to
// synchronization issues.

// 'critical' specifies that code is
// executed by only one thread at a
// time i.e., only one thread enters
// the critical section at a given time
#pragma omp critical

    for (i = 1; i > 0; i++) {

        printf("\nEnter your choice:");
        scanf("%d", &n);

        // Switch Cases
        switch (n) {
        case 1:

            // If mutex is 1 and empty
            // is non-zero, then it is
            // possible to produce
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }

            // Otherwise, print buffer
            // is full
            else {
                printf("Buffer is full!");
            }
            break;

        case 2:
```

```
        // If mutex is 1 and full
        // is non-zero, then it is
        // possible to consume
        if ((mutex == 1)
            && (full != 0)) {
            consumer();
        }

        // Otherwise, print Buffer
        // is empty
        else {
            printf("Buffer is empty!");
        }
        break;

    // Exit Condition
    case 3:
        exit(0);
        break;
    }
  }
}
```

Output:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:2
Buffer is empty!
Enter your choice:1

Producer producesitem 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:3


=== Code Execution Successful ===
```

Conclusion:

What is Semaphore?

A semaphore is a synchronization primitive used in concurrent programming to control access to shared resources. It is essentially a counter that is used to manage access to a finite set of resources, such as a critical section of code, a shared memory buffer, or a shared data structure. Semaphores can be thought of as signaling mechanisms that allow threads or processes to coordinate their activities and avoid race conditions, where multiple entities attempt to access or modify the same resource simultaneously.

There are two types of semaphores: binary semaphores and counting semaphores. Binary semaphores, also known as mutexes (short for mutual exclusion), have two states: locked (1) and unlocked (0). They are typically used to protect critical sections of code, allowing only one thread or process to access the protected resource at a time. Counting semaphores, on the other hand, can have a value greater than one and are used to control access to multiple instances of a resource. Threads or processes can increment or decrement the value of a counting semaphore, depending on whether they are acquiring or releasing the resource.

What are different types of Semaphore?

Semaphores are synchronization primitives used in concurrent programming to control access to shared resources. They come in different types, each with its own characteristics and use cases.

1. **Binary Semaphores**: Also known as mutexes (mutual exclusion semaphores), binary semaphores are the simplest type. They have only two states: locked (1) and unlocked (0). Binary semaphores are typically used to protect critical sections of code or to synchronize access to a single resource between multiple threads or processes. They ensure that only one thread or process can access the resource at a time.

2. **Counting Semaphores**: Unlike binary semaphores, counting semaphores can have a value greater than 1. They allow multiple threads or processes to access a shared resource simultaneously, up to a specified maximum limit. Counting semaphores are often used to control access to a finite pool of identical resources, such as a fixed-size buffer or a pool of worker threads. Each time a resource is acquired, the semaphore's value is decremented, and it's incremented when the resource is released.

3. **Named Semaphores**: Named semaphores are semaphores that are associated with a unique name within the operating system's namespace. They allow unrelated processes to synchronize access to shared resources by using the same named semaphore. Named semaphores are commonly used for inter-process communication (IPC), where multiple processes need to coordinate their activities or share data.

4. **Unnamed Semaphores**: Also referred to as anonymous semaphores, unnamed semaphores are not associated with a specific name. They are typically used for synchronization between threads within the same process. Unnamed semaphores are created dynamically within the program and are often implemented as variables within shared memory or as part of a synchronization primitive provided by threading libraries.

5. **Binary vs. Counting Semaphores**: While binary and counting semaphores serve similar purposes, they differ in their usage scenarios. Binary semaphores are well-suited for protecting critical sections and implementing mutual exclusion, where only one entity should access a resource at a time.